The client program should create a client socket — an instance of the class `Socket` — pointing at the computer/port on which the `ServerSocket` is running. The client program can connect to the server using hostnames or IP addresses too, for example:

```
clientSocket = new Socket("124.67.98,101", 3000);
clientSocket = new Socket("localhost", 3000);
clientSocket = new Socket("127.0.0.1", 3000);
```

While deciding which port number to use for the `ServerSocket`, avoid using port numbers below 1024 to avoid conflicts with other system programs. For example, port 80 is typically used by HTTP servers; port 443 is reserved for HTTPS; port 21 is typically used for FTP communications; port 389 is for LDAP servers, and so on. After creating a socket-based connection, both client and server should obtain references to its input/output streams and use them for data exchange.

Why Use Sockets?

Why even use manual socket programming if you can easily establish inter-computer communication with, say, HTTP-based protocol (it uses sockets internally), start one of many open-source or commercial web servers, and have clients connect to the server as shown in the preceding sample programs in this lesson? Because a socket connection has a lot less overhead than any standard protocol.

You can create your own very compact protocol that will allow you to send only the data you need, with no or minimal headers. Socket communication provides a duplex byte stream, whereon the data travels simultaneously in both directions, unlike protocols based on the request-response model. Think of financial trading systems: Speed is the king there, and the ability to send data up and down at the same time saves milliseconds, which makes a difference.

On the other hand, if you design your application to use sockets, the live connection will be maintained for each user connected to `ServerSocket`. If your program has to maintain several thousand concurrent connections it will require more powerful servers than programs using the request-response system, with which a connection is maintained only during the time of the client's request.

The Stock Quote Server with Sockets

Let's build a socket-based client/server application that emulates both a server providing fake price quotes for requested stocks and a client consuming this data.

The server will start and listen to requests on port 3000 (see Listing 18-5). The method `accept()` of the `SocketServer` class is the one that waits for the client's requests. As soon as it starts you'll see the message "Waiting for a quote request" on the system console, and nothing else will happen until the request comes in from the client. Creating a `SocketServer` instance binds it to the specified port, but if this port is already in use by another process you'll get a `BindException`.



LISTING 18-5: The server generating stock quotes

```
import java.io.*;
import java.net.*;
public class StockQuoteServer {
```

```
public static void main(java.lang.String[] args) {
  ServerSocket serverSocket = null;
  Socket client = null;

  BufferedReader inbound = null;
  OutputStream outbound = null;

  try
                {
                // Create a server socket
                serverSocket = new ServerSocket(3000);

                 System.out.println("Waiting for a quote request");
                while (true)
                {
                  // Wait for a  request
                  client = serverSocket.accept();

       // Get the streams
                   inbound=new BufferedReader(new
          InputStreamReader(client.getInputStream()));
                 outbound =
                              client.getOutputStream();

                String symbol = inbound.readLine();

       //Generete a random price
       String price= (new
                  Double(Math.random()*100)).toString();

       outbound.write(("\n The price of "+symbol+
                    " is " + price + "\n").getBytes());

                     System.out.println("Request for " + symbol +
            " has been processed. The price is " + price);
        outbound.write("End\n".getBytes());
                }
  }
  catch (IOException ioe)
 {
                System.out.println("Error in Server: " + ioe);
 } finally{
     try{
         inbound.close();
         outbound.close();
       }catch(Exception e){
             System.out.println("StockQuoteServer: can't close streams"
                                                    + e.getMessage());
       }
   }
 }
}
}
```

When a client connects to the socket, the server gets references to its I/O streams and sends randomly generated quotes for the requested stock. In the real world this server would have to be connected to another program providing real-time market data, but for our purposes generating random numbers as "price quotes" will suffice. The client program shown in Listing 18-6 has to be started with a command-line parameter such as `IBM` or `MSFT` to produce a price quote.



LISTING 18-6: The client sending requests for stock quotes

```
import java.io.*;
import java.net.*;


public class Client {

public static void main(java.lang.String[] args) {

  if (args.length==0){

   System.out.println("Usage: java Client Symbol");
          System.exit(0);
 }

 OutputStream outbound = null;
 BufferedReader inbound = null;

 Socket clientSocket = null;

   try {

      // Open a client socket connection
      clientSocket = new Socket("localhost", 3000);

      System.out.println("Client: " + clientSocket);

      // Get the streams

        outbound = clientSocket.getOutputStream();
        inbound=new  BufferedReader(new
         InputStreamReader(clientSocket.getInputStream()));

      // Send stock symbol to the server
      outbound.write((args[0]+"\n").getBytes());
      outbound.write("End\n".getBytes());

      String quote;
      while (true){
        quote = inbound.readLine();
        if (quote.equals("End")){
                break;
         }
        System.out.println("Got the quote for " + args[0]+":" + quote);
       }
```

```
        }catch (UnknownHostException uhe){

          System.out.println("UnknownHostException: " + uhe);

        } catch (IOException ioe){

           System.err.println("IOException: " + ioe);
        } finally{

         // Close the streams
         try{
             outbound.close();
             inbound.close();
             clientSocket.close();
         } catch(IOException e){
              System.out.println("Can not close streams..." +
                                                 e.getMessage());
         }
        }
    }

}
```

Have you noticed that `StockQuoteServer` appends the word `End` to indicate that the price quote has ended? The class `Client` also checks for this word. This is an example of a very simple custom-made networking protocol.

TRY IT

The goal of this exercise is to test the socket communication in action, even if you have only one computer.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson18 folder in the download.*

Hints

In this exercise you'll be using two separate command windows to run the client and the server. But Eclipse IDE also enables you to have more than one Console view. Find a little icon that looks like a monitor in the Console view toolbar and click the little triangle next to it to switch between toolbars.

Step-by-Step

1. Create a new Eclipse project called Lesson18. Copy into this project and compile the classes `StockQuoteServer` and `Client` from the DVD accompanying the book.
2. Even though you can run both programs from Eclipse, it's easier to observe the entire process if you run them from separate command windows. Open two command windows and imagine that they belong to different computers.
3. In each command window, go to the bin directory of the Lesson 18. In one of them start the server and in the other start the client — for example

```
java StockQuoteServer
java Client IBM
```

4. Observe that the server generates prices, and that both client and server print the same price. By starting client and server in different command windows you are starting two separate JVM's, emulating network communication between computers.
5. Open a couple more command windows and start the client program in them, providing different stock symbols as arguments. Observe that the same server can handle multiple clients' requests. If you have access to a real network in which each computer has Java run time installed, run the client and server programs on different computers — just replace the `localhost` in the class `Client` with the network name or IP address of the server's computer.



Please select Lesson 18 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

19

Processing E-Mails with Java

People send and receive e-mails using e-mail client programs such as Microsoft Outlook and Gmail. Can you write your own program to send and receive e-mails? The chances are slim that you'll need to develop a client to compete with Gmail, but enterprise Java applications often need the ability to send or receive e-mails in certain conditions.

For example, an application that processes credit card transactions suspects a fraudulent activity and needs to send an e-mail to the account owner. Such an application becomes an e-mail client. A Java application provided by your bank can run on your Android smartphone and have an integrated e-mail inbox that accumulates all messages sent to you by the bank.

Another good example involves the forgotten-password functionality. A user clicks the Forgot Your Password? hyperlink and enters a valid e-mail address in a text input field. A couple of seconds later an e-mail containing instructions for resetting the password arrives in her e-mail inbox. The server-side Java program sent this e-mail automatically using JavaMail's *application program interface (API)*.

In this lesson you'll learn how to write a Java program that can send e-mails reminding you about your friends' birthdays.

PROTOCOLS AND SERVERS

For e-mail messaging you'll be using standard protocols: *Simple Mail Transfer Protocol (SMTP)* for sending e-mails and *Post Office Protocol (POP)* or *Internet Message Access Protocol (IMAP)* for retrieving e-mails. The third version of this protocol is called *POP3*; IMAP is a more advanced alternative to POP3. The format of the message body and its attachments, if any, is defined by the *Multipurpose Internet Mail Extensions (MIME)* protocol.

Some vendors use proprietary protocols for e-mails, but we'll stick to standards. For Java SE, JavaMail API is an optional package that has to be downloaded separately, but it is included

with Java EE. The API is simple to use — it takes only a few lines of code to send an e-mail from a Java program.

To test a client program that sends e-mails you'll need to know the hostname and port of some SMTP server and have an account with the service provider. Typically such servers use port 25 or 587 for e-mails encrypted using *Transport Layer Security (TLS)* protocol and port 465 for *Secure Sockets Layer (SSL)* encryption. Some ISPs may block these ports though.

Apart from an SMTP server hosted by your ISP, you can use the free public server `smtp.gmail.com` offered by Google. For the parameters of this server refer to the following Web page: `http://mail.google.com/support/bin/answer.py?hl=en&answer=13287`. Yahoo! also offers a free public SMTP server at `smtp.mail.yahoo.com`.

Here's a word of caution: After you learn how easy it is to write a Java program that sends e-mails, you might have an urge to start sending mass e-mails to large lists of recipients to promote your business or share the latest jokes you've found online. Unsolicited e-mail is called spam, and if you write a program that sends dozens of e-mails per minute your ISP will probably notice it and give you a warning or even terminate your Internet access.

CREATING BIRTHDAY MAILER

Because I'll be writing a birthday mailer program using Java SE, I need to download the *jars* (libraries) with JavaMail classes. At the time of this writing JavaMail API 1.4.3 is the latest release, and you can download it from `www.oracle.com/technetwork/java/index-138643.html`.

Unzip the JavaMail archive — it includes all required jar files, demos, and documentation. Create an Eclipse project called Lesson19, right-click the project name, and select Properties ➪ Java Build Path. Go to the Libraries tab, click Add External JARs, and add `mailapi.jar` and `smtp.jar` to the project from the `lib` directory of the extracted archive. These jars contain all the classes from the `javax.mail` package required to send e-mails. If you are using a really old version of Java (prior to version 6) you'll also need to download and add to your project the JavaBeans Activation Framework (JAF).

Required Supporting Classes

At a minimum you have to use the following classes to send an e-mail:

- `javax.mail.Session`: This class knows important characteristics of your environment, such as the name of the mail server.
- `javax.mail.Message`: This class represents your message, containing recipients, subject, content, and so on.
- `javax.mail.internet.InternetAddress`: The recipient of your message will be represented by this class.
- `javax.mail.Transport`: This class will pass your message for the delivery to a protocol-specific transport.

Page 236 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

Writing the Mail Sender

Now you'll learn how to write a program to send e-mail messages. I'll be using the publicly available SMTP server from Yahoo!, but if you decide to use the server provided by your ISP you can find out its hostname and port from the settings of your current e-mail client program. For example, Microsoft Outlook has this information in the menu Tools ➪ Accounts ➪ Mail ➪ Properties.

Typically SMTP servers require you to provide a valid e-mail address and password for authentication. Because your JavaMail program won't use a log-on window, you can hard-code e-mail credentials in a subclass of JavaMail's `Authenticator`, as shown in Listing 19-1.



LISTING 19-1: Authenticator class

```
package com.practicaljava.lesson19;

import javax.mail.Authenticator;

import javax.mail.PasswordAuthentication;

class MyMailAuthenticator extends Authenticator {

    protected PasswordAuthentication getPasswordAuthentication() {

      return new PasswordAuthentication ("test@yahoo.com", "password123");

    }
}
```

If `MyMailAuthenticator` isn't able to authenticate using the provided e-mail/password combination, it'll throw the `AuthenticationFailedException` with the error message "Access Denied." If you'll specify a nonexistent sender's e-mail address, the program will throw `SMTPSendFailedException` with the error "From address not verified." If authentication is not required, the class `MyMailAuthenticator` is not needed.

A sample file that stores the birthdays of your e-mail contacts is shown in Listing 19-2. It's a file in a CSV format, where each line contains the birthday, name, and e-mail address of a contact.



LISTING 19-2: The file birthday.txt

```
Aug-9,Mary Lou,mlou@gmail.com
Mar-31,John Smith,jsmith@gmail.com
Aug-9,David Lee,dlee@yahoo.com
```

The `Mailer` program (see Listing 19-3) will read this file into an array, and if the birthday's date is the same as the current date, it'll send a "happy birthday" e-mail to this person. The program starts by creating an object with the properties of the SMTP server at Yahoo! Then it creates an instance of the `MyMailAuthenticator` class and gives both of them to the `Session` instance.

When this is done the method `main()` invokes the method `readBirthdayFile()`, which populates `ArrayList friends` with the data from the file `birthday.txt`. Then the method

`iterateThroughBirthdays()` parses only those lines in which the birthday date is the same as the current date. This program uses the class `java.util.Scanner`, which parses the content of the String (the friend's data) based on specified delimiters.

The `setPropsAndSendEmail()` method has the code for sending e-mails. First it creates an instance of the `Message` object and populates it with all the familiar parameters: from, recipient(s), subject, data, and the text of the message. The call to `Transport.send()` performs the actual process of sending this message.



LISTING 19-3: The Mailer class

```
package com.practicaljava.lesson19;

import java.io.*;
import java.text.*;
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;

public class Mailer      {

    private Session session = null;

    private static  String emailSenderAddress = "yfain11@yahoo.com";
    private static  String emailSubject = "Happy Birthday!!!";
    private String emailText = "Happy birthday dear %s!!!";

    ArrayList<String> friends = new ArrayList<String>();

    Mailer() {

        Properties sessionProperties = new Properties();
        sessionProperties.put("mail.smtp.host", "smtp.mail.yahoo.com");
        sessionProperties.put("mail.smtp.user", emailSenderAddress);
        sessionProperties.put("mail.smtp.port", "25");
        sessionProperties.put("mail.smtp.auth", "true");

        MyMailAuthenticator authentificatorForMessage =
                                            new MyMailAuthenticator();

        session = Session.getInstance(sessionProperties,
                                           authentificatorForMessage);
    }



    private void setPropsAndSendEmail(String emailRecipient, String emailText){

        try{

            Message emailMessage = new MimeMessage(session);

            emailMessage.setFrom(new InternetAddress(emailSenderAddress));
```

```
            emailMessage.setRecipients(Message.RecipientType.TO,
                                InternetAddress.parse(emailRecipient, false));
            emailMessage.setSubject(emailSubject);
            emailMessage.setSentDate(new Date());
            emailMessage.setText(emailText);
            Transport.send(emailMessage);
            System.out.println("Your email to " + emailRecipient +
                                            " has been sent successfully");
        }catch(Exception e){

            System.out.println("Your email to " + emailRecipient +
                               "  has not been sent: " + e.getMessage());
            e.printStackTrace();
        }
    }


    private void readBirthdayFile() throws IOException {

        FileInputStream birthdayFile = new FileInputStream("birthday.txt");

        BufferedReader birthdayFileReader = new BufferedReader(
                                        new InputStreamReader(birthdayFile));

        String friendInfo;

        while ((friendInfo = birthdayFileReader.readLine()) != null){
            friends.add(friendInfo);
        }

        birthdayFileReader.close();

        birthdayFile.close();
    }


    private void iterateThroughBirthdays(){

        Iterator<String> iterator = friends.iterator();

        while (iterator.hasNext()){

            scanForManInfoAndSendEmail(iterator.next());

        }
    }


    private void scanForManInfoAndSendEmail(String stringFromArray){

        Scanner scannerOfLines = new
```

continues

LISTING 19-3 *(continued)*

```
                    Scanner(stringFromArray).useDelimiter("[,\n]");

        if (scannerOfLines.next().equals(getCurrentDateMMMd())) {

            String emailAddressee = scannerOfLines.next();
            String emailAddress = scannerOfLines.next();
            setPropsAndSendEmail(emailAddress,
                        String.format(emailText, emailAddressee));
        }
    }

    private static String getCurrentDateMMMd(){

        return new SimpleDateFormat("MMM-d", Locale.US).format(new
                                    GregorianCalendar().getTime());

    }


    public static void main(String[] args){

        Mailer mm = new Mailer();

        try {

            mm.readBirthdayFile();
            mm.iterateThroughBirthdays();

        } catch (IOException e) {

            e.printStackTrace();
        }
    }
}
```

HOW TO RETRIEVE E-MAILS

Even though more often than not the receivers of the messages sent by a Java application will use one of the popular e-mail clients already available, you may need to incorporate message-retrieving functionality into your application. In this section I'll just give an overview of the process of retrieving e-mails programmatically, and in the "Try It" section you'll write such a program on your own.

As in your local post office, incoming mail should be stored somewhere. The POP3 or IMAP mail server provides such storage. POP3 is an older protocol that retrieves messages from a single mailbox. IMAP is newer and more advanced: It allows the organization of messages in folders with filtering support.

To get access to the user's incoming e-mails the Java program needs to connect to the mail server and get an instance of the `Store` object by calling the method `getStore()` on the `Session` object.

You'll need to know the hostname of your mail server, the protocol it uses, and the relevant user ID and password in order to get the `Store` instance.

The next step is establishing a connection with `Store` by calling its method `connect()`. After this is done, get `store`'s default root folder and then a specific sub-folder, such as `INBOX`.

Folders in JavaMail are represented by the class `Folder`. Its method `open()` will open the specified folder for `READ_ONLY` or `READ_WRITE`. Finally, the `getMessages()` method returns an array of `Message` objects. Iterating through this array will allow you to retrieve message data by calling `getSubject()`, `getFrom()`, `getContent()`, and so on.

Listing 19-4 shows a fragment of the code for retrieving messages from a Gmail account. To enable or disable the IMAP or POP protocol, go to Setting ➪ Forwarding POP/IMAP in your Gmail client.



LISTING 19-4: Fragment of code for e-mail retrieval from Gmail

```
Properties properties = System.getProperties();
Session session = Session.getDefaultInstance(properties);

String host = "pop.gmail.com"; // or "imap.gmail.com"
String username = "yakov";
String password = "mypasswd";
Store store = session.getStore("pop3s"); // or "imap"
store.connect(host, username, password);

Folder rootFolder = popStore.getDefaultFolder();
defaultPopFolder = defaultPopFolder.getFolder("INBOX");
defaultPopFolder.open(Folder.READ_ONLY);


Message[] message = folder.getMessages();

for (int i = 0; i < message.length; i++) {

     // get message parts here
}
```

JavaMail has full support for multipart messages with attachments (see the `Multipart`, `MimeMultipart`, and `MimeBodyPart` classes), but multipart message support is beyond the scope of this book. The description of these classes is available in the Java help system. Look for these class descriptions in the package `javax.mail`.

USEFUL OPEN-SOURCE COMPONENTS

Apache Commons is a community project that offers a number of open-source reusable components. One of the libraries contains classes for sending and receiving e-mails form Java. You can read about this project at `http://commons.apache.org/email/userguide.html`.

The process of sending a simple e-mail with this library comes down to creating an instance of one class, `SimpleEmail`. Then you need to set the properties of this object: host, port, from, to, authenticator, subject, and message body. When this is done, just call the `send()` method.

If all you need is to send simple e-mails, the benefits of using the Apache Commons e-mail library may not be so obvious. But if your program needs to send multipart e-mails with attachments, or HTML-formatted e-mails, consider using this library. One useful feature of this library is that it gives you the ability to specify an alternative recipient for bounced e-mails.

There is one more community project to consider: Apache Velocity (`http://velocity.apache.org/`). Imagine that you need to write a program that will perform mass mailing of pre-formatted letters, inserting customer-specific information (such as name and address) into each one. The Apache Velocity project would be quite handy for this purpose. It is also useful if you want to send automatic account registrations or password-change e-mails. Instead of hard-coding the letter text into your Java program, you can create text files (templates) containing special markup to be substituted with real values by your Java program during run time.

For further details, read the user guide at the following URL: `http://velocity.apache.org/engine/devel/user-guide.html`.

Third-party libraries (e.g. Spring Framework, discussed in Lesson 35) hide specifics of the use of JavaMail API. Check out the documentation on using e-mails with the Spring Framework at the following URL: `http://static.springsource.org/spring/docs/2.0.x/reference/mail.html`.

TRY IT

Write a Java program that will retrieve messages from your mail inbox and print their subjects on the system console.

Lesson Requirements

You should have Java installed. For this section I assume that you've created Eclipse project Lesson19 and that it contains a working version of the `Mailer` program.



You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. You can find them in the Lesson19 folder in the download.

Step-by-Step

1. Add JavaMail libraries `pop.jar` and `imap.jar` to the project.
2. Read the chapter on message storage and retrieval in the document "JavaMail Design Specification" and code sample "Showing the message" from Listing B-X in Appendix B of "JavaMail Design Specification." This document is located in the `doc` folder of the JavaMail distribution.

3. Write the Java program `MailReader` to retrieve and print on the system console the subjects of messages located in your inbox.
4. Run `MailReader` and compare the printed message subjects with the ones displayed by the mail client you use regularly.



Please select Lesson 19 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

20

Introduction to Multi-Threading

In March of 2010 I was interviewed by Janice Heiss from Oracle (`http://java.sun.com/developer/technicalArticles/Interviews/community/yakov_qa_2010.html`), and she asked me to name the Java class I couldn't live without. I didn't think twice: `Thread`. This is where the power of Java resides. There are books devoted to Java threads, which give you amazing power and flexibility when it comes to building highly available, scalable, and responsive applications capable of processing thousands of concurrent requests of various types.

Up until now you've been creating Java programs that were executing code sequentially. But the main power of Java lies in its ability to do things in parallel, or, as they say, to run *multiple threads of execution*. As always, going through a practical use case is the best way to understand this feature.

Let's discuss a program that should display market news and information about the user's stock portfolio in the same window. While the market news feed is coming from a remote computer, stock portfolio data are retrieved from the database and may be located on the local machine.

Suppose it takes five seconds to get the market news and only three seconds to get the portfolio data. If you run these two tasks sequentially (one after another), you need eight seconds to complete the job.

But market news doesn't depend on your portfolio data and these two tasks can run in parallel. They run on different computers and use different processors. If you can implement parallel processing, the total time should be less than eight seconds (close to five seconds in our use case — the time it takes for the longer task to complete).

A Java program can start multiple threads of execution that will run in parallel. Even if you have only one processor in your computer, it still could be a good idea to run some tasks in parallel. Think of a web browser that allows you to download a file and perform page browsing at the same time. Web browsers maintain two or more connections to any website, and can download multiple resources (text, images, videos, music) at the same time. Any web browser works in a multi-threaded mode.

If these jobs ran sequentially, the browser's window would be frozen until the download was complete. On a multiprocessor computer, parallel threads can run on different CPUs. On a single-processor computer, threads will take turns getting "slices" of the processor's time. Because switching CPU cycles between threads happens fast, a user won't notice the tiny delays in each thread's execution, and browsing will feel smooth.

In many cases, especially on single CPU computers, the benefit of many threads comes about because there's a lot of idle time in most operations. In particular, if an operation is I/O bound instead of CPU bound using multiple threads helps take advantage of those otherwise unused blocks of time.

People also can work in a multi-threaded mode. For example, they can drink coffee while talking on a cell phone and driving a car. In the airport you can see people walking on the moving belt — they will reach their gate faster than by standing still.

THE CLASS THREAD

If class `A` needs to initiate some executions in classes `B` and `C`, the latter two must declare multi-threading support from the get-go. Each of the classes `B` and `C` must either be inherited from the Java class `Thread` or implement one of the following interfaces: `Runnable` or `Callable` (the latter is covered in Lesson 21). If a class is inherited from the class `Thread` it has to override the method `run()`.

The first version of the market-portfolio example has three classes (see Listings 20-1, -2, and -3). Two of them are subclasses of the class `Thread` (`MarketNews` and `Portfolio`), and the third (`TestThreads`) is just a testing program that instantiates them and starts the execution of some code in each of them. You must initiate the code that has to work as a thread in the method `run()`.



LISTING 20-1: Class MarketNews

```
public class MarketNews extends Thread {
 public MarketNews (String threadName) {
        super(threadName);  // name your thread
    }

  public void run() {
      System.out.println( "The stock market is improving!");
     }
}
```



LISTING 20-2: Class Portfolio

```
public class Portfolio extends Thread {
    public Portfolio (String threadName) {
        super(threadName);
    }

    public void run() {
       System.out.println( "You have 500 shares of IBM ");
     }
}
```



LISTING 20-3: Class TestThreads starts MarketNews and Portfolio

```
public class TestThreads {
    public static void main(String args[]){
      MarketNews mn = new MarketNews("Market News");
      mn.start();

      Portfolio p = new Portfolio("Portfolio data");
      p.start();

      System.out.println( "TestThreads is finished");
    }
}
```

The method `main()` in Listing 20-3 instantiates each thread, passing the name for the thread as a constructor argument, and then calls its `start()` method. Each thread itself invokes internally the code located in its method `run()`. After calling `mn.start()`, the program `TestThread` does not wait for its completion but immediately executes the lines that follow, creating and starting the thread `Portfolio`. Even if `MarketNews` will take time, the `Portfolio` thread will start immediately.

If you run the `TestThread` program it'll print the output from threads `MarketNews` and `Portfolio` almost simultaneously — there is no lengthy and time-consuming code in their `run()` methods. A bit later, in the section "Sleeping Threads," I'll show you how to emulate a lengthy execution. The output of the `TestThread` program can vary — it all depends on which thread will finish first.

THE INTERFACE RUNNABLE

The second way to create threads is to implement a `Runnable` interface. In this case your class also has to have business logic in the method `run()`. The second version of our market-portfolio example (Listings 20-4, -5, and -6) also has three classes, but `MarketNew2` and `Portfolio2` are not inherited from the class `Thread` — they implement the `Runnable` interface.

Creation of a thread in this case is a two-step process: create an instance of a class that implements `Runnable` and then give it as a constructor argument during instantiation of the class `Thread`.



LISTING 20-4: Class MarketNews2

```
public class MarketNews2 implements Runnable {
     public void run() {
        System.out.println( "The stock market is improving!");
     }
}
```



LISTING 20-5: Class Portfolio2

```
public class Portfolio2 implements Runnable {
    public void run() {
       System.out.println( "You have 500 shares of IBM ");
     }
}
```



LISTING 20-6: Class TestThreads2

```
public class TestThreads2 {
    public static void main(String args[]){

     MarketNews2 mn2 = new MarketNews2();
     Thread mn = new Thread(mn2,"Market News");
     mn.start();

     Runnable port2 = new Portfolio2();
     Thread p = new Thread(port2, "Portfolio Data");
     p.start();

     System.out.println( "TestThreads2 is finished");
    }
}
```

Note that I've declared the variable `port2` in Listing 20-6 to be not of type `Portfolio2`, but of type `Runnable`. I did it for illustration purposes and to encourage you to use what you've learned about casting to interfaces in Lesson 7 (see Listing 7-5). It takes three lines of code in Listing 20-6 to instantiate and start a thread. You can do this in one line, but this syntax won't give you a reference to the thread object should you need to call any methods on this object in the future (see variables `mn` and `p` in Listing 20-6):

```
(new Thread(new MarketNews2("Market News"))).start();
```

The `Runnable` interface provides a more flexible way to use threads, because it allows a class to be inherited from any other class, while having all the features of a thread. For example, a Swing applet must be a subclass of a `javax.swing.JApplet`, and, because Java doesn't support multiple inheritance, it should implement `Runnable` to support multi-threading.

SLEEPING THREADS

One of the ways to make the processor available to other threads is by using `Thread`'s method `sleep()`, which enables you to specify in milliseconds (and nanoseconds) how long the thread has to sleep. For example, the thread `MarketNews3` in Listing 20-7 puts itself to sleep for a thousand milliseconds (one second) after each output of the message about market improvements.



LISTING 20-7: Class MarketNews3

```
public class MarketNews3 extends Thread {
   public MarketNews3 (String str) {
    super(str);
   }

  public void run() {
    try{
      for (int i=0; i<10;i++){
```

```
            sleep (1000);  // sleep for 1 second
            System.out.println( "The market is improving " + i);
           }
         }catch(InterruptedException e ){
            System.out.println(Thread.currentThread().getName()
                                          + e.toString());
         }
       }
    }
```

When `MarketNews3` goes to sleep, the thread `Portfolio` from Listing 20-8 gets the CPU and will print its message and then sleep for 700 milliseconds on each loop iteration. Every second `MarketNews3` wakes up and does its job.



LISTING 20-8: Class Portfolio3

```
public class Portfolio3 extends Thread {
   public Portfolio3 (String str) {
        super(str);
   }

   public void run() {
     try{
      for (int i=0; i<10;i++){
       sleep (700);    // Sleep for 700 milliseconds
       System.out.println( "You have " +  (500 + i) +
                                       " shares of IBM");
      }
    }catch(InterruptedException e ){
      System.out.println(Thread.currentThread().getName()
                                          + e.toString());
    }
   }
}
```

After adding the sleeping part to our thread, the program TestThreads3 (see Listing 20-9) will generate mixed console output about market and portfolio from both threads — this proves that they are taking turns even with the single-processor machine.



LISTING 20-9: Class TestThreads3

```
public class TestThreads3 {

    public static void main(String args[]){

         MarketNews3 mn = new MarketNews3("Market News");
         mn.start();

         Portfolio3 p = new Portfolio3("Portfolio data");
```

continues

LISTING 20-9 *(continued)*

```
            p.start();

            System.out.println( "The main method of TestThreads3 is finished");
        }
    }
```

If you need to "wake up" a sleeping thread before its sleeping time is up, use the method `interrupt()`. Just add `mn.interrupt()` to the class `TestThreads3` right after starting the `MarketNews` thread. This will trigger `InterruptedException`, and `MarketNews3` will wake up and continue its execution from the operator located below the `sleep()` method call. The class `Thread` has a method `interrupted()` that returns `true` if the current thread has been interrupted. The output of the program `TestThreads3` can look as follows:

```
The main method of TestThreads3 is finished
You have 500 shares of IBM
The market is improving 0
You have 501 shares of IBM
The market is improving 1
You have 502 shares of IBM
You have 503 shares of IBM
The market is improving 2
You have 504 shares of IBM
The market is improving 3
You have 505 shares of IBM
You have 506 shares of IBM
The market is improving 4
You have 507 shares of IBM
The market is improving 5
You have 508 shares of IBM
The market is improving 6
You have 509 shares of IBM
The market is improving 7
The market is improving 8
The market is improving 9
```

These days it's hard to find a single-CPU server machine. Most of the readers of this book have dual-core CPUs — these have two processors in the same chip. Modern JVMs use multiple cores for multi-threaded applications, but you shouldn't assume that your program will run twice as fast on such hardware. JVM optimization is a complex subject and is out of the scope of this tutorial. You may boost the performance of your system by increasing the number of threads running in parallel, but you should define the right ratio between the number of threads and the number of processors during the performance-tuning phase of application development.

HOW TO KILL A THREAD

The class `Thread` has a method, `stop()`, that was supposed to know how to kill the current thread. But it was deprecated many years ago because it could bring some of the objects in your program into an inconsistent state caused by improper locking and unlocking of the object instances.

There are different approaches to killing threads. One of them involves creating your own method on the thread, say `stopMe()`, in which you set your own `boolean` variable, say `stopMe`, to `false` and test its value periodically inside the thread's method `run()`. If application code will set the value of `stopMe` to `true`, just exit the code execution in the method `run()`. In Listing 20-10, the loop in the method `run` checks the value of the variable `stopMe`, which is initialized with the reference to the current `Thread` object. As soon as it is changed (set to `null` in this case), the processing will complete.



LISTING 20-10: Killing a thread

```
class KillTheThread{
           public static void main(String args[]){
           Portfolio4 p = new Portfolio4("Portfolio data");
         p.start();

         // Some other code goes here,
         // and now it's time to kill the thread
         p.stopMe();

           }
}

class Portfolio4 extends Thread{

private volatile Thread stopMe = Thread.currentThread();

    public void stopMe() {
        stopMe = null;
    }

    public void run() {
        while (stopMe == Thread.currentThread()) {
          try{
            //Do some portfolio processing here
          }catch(InterruptedException e ){
            System.out.println(Thread.currentThread().getName()
                                    + e.toString());
        }

     }
 }
```

The variable `stopMe` has been declared with a `volatile` keyword, which warns the Java compiler that another thread can modify it and that this variable shouldn't be cached in registers, so that all threads must always see its fresh value. The class Portfolio4 could be written differently — the variable `stopMe` could be declared as `boolean`.

Not every thread can be killed using the code shown in Listing 20-10. What if a thread is not doing any processing at the moment, but is waiting for the user's input? Call the method `interrupt()` on such a thread. Killing a thread by interrupting it may be the only technique you need to use in such cases.

If you need to kill a thread that's busy doing some blocking I/O operations and the preceding methods of killing such a thread don't work, try closing I/O streams — this will cause `IOException` during the current read/write operation and the thread will be over.

If you'd like to read more comprehensive coverage of this subject, see Dr. Heinz Kabutz the Java Specialist's newsletter Issue #56, available at `www.javaspecialists.co.za/archive/Issue056.html`.

THREAD PRIORITIES

Single-processor computers use a special scheduling algorithm that allocates processor time slices to the running threads based on their priorities. If `Thread1` is using the processor and the higher-priority `Thread2` wakes up, `Thread1` is pushed aside and `Thread2` gets the CPU. It is said that `Thread2` *preempts* `Thread1`.

The class `Thread` has a method, `setPriority()`, that allows you to control its priority. There are 10 different priorities, which are `final integer` variables defined in the class `Thread`. Some of them are named constants `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`. Here's an example of their usage:

```
Thread myThread = new Thread("Portfolio");
myThread.setPriority(Thread.NORM_PRIORITY + 1);
```

If two threads with the same priority need the processor, it'll be given to one of them using an algorithm specific to the computer's OS.

THREAD SYNCHRONIZATION AND RACE CONDITIONS

During the design stage of a multi-threaded application's development you should consider the possibility of a so-called *race condition*, which happens when multiple threads need to modify the same program resource at the same time (*concurrently*). The classic example is when a husband and wife are trying to withdraw cash from different ATMs at the same time.

Suppose the balance on their joint account is $120. If a `Thread` class is responsible for the validation and update of the balance of their bank account, there is a slight chance that the husband's thread will enable him to withdraw $100, but that before the actual withdrawal the wife's thread will come in trying to validate a $50 withdrawal. She is also enabled to make her withdrawal, because $120 is still there! The couple would successfully withdraw a total of $150, leaving a negative balance in their account. This is an example of a race condition.

A special keyword, `synchronized`, prevents race conditions from happening. This keyword places a *lock* (a monitor) on an important object or piece of code to make sure that only one thread at a time will have access. The code in Listing 20-11 locks the entire method `withdrawCash()` so no other thread will get access to the specified portion of code until the current (locking) thread has finished its execution of `withdrawCash()`.



LISTING 20-11: Declaring a synchronized method

```
class ATMProcessor extends Thread{
   ...
 synchronized withdrawCash(int accountID, int amount){
    // Some thread-safe code goes here, i.e. reading from
    // a file or a database
      ...
   boolean allowTransaction = validateWithdrawal(accountID,
                                                 amount);
   if (allowTransaction){
     updateBalance(accountID, amount, "Withdraw");
   }
   else {
    System.out.println("Not enough money on the account");
   }
  }
}
```

The locks should be placed for the shortest possible time to avoid slowing down the program: That's why synchronizing short blocks of code is preferable to synchronizing whole methods. Listing 20-12 shows how to synchronize only the portion of the code that may cause the race condition, rather then locking the entire method `withdrawCash()`.



LISTING 20-12: Declaring a synchronized block

```
class ATMProcessor extends Thread{
  ...
 withdrawCash(int accountID, int amount){
    // Some thread-safe code goes here, i.e. reading from
    // a file or a database
   ...
  synchronized(this) {
   if (allowTransaction){
    updateBalance(accountID, amount, "Withdraw");
   }
   else {
    System.out.println("Not  enough money on the account");
   }
  }
 }
}
```

When a `synchronized` block is executed, the section of the code in parentheses is locked and can't be used by any other thread that's locked on the same section of the code until the lock is released. Listing 20-12 locks the current instance of the class `ATMProcessor` (represented by the `this` keyword) only for the duration of the `updateBalance()` method, which is a shorter period of time than locking `withdrawCash()` would take.

JAVA ENUMERATIONS

The `enum` construct is not related to the thread concept and can be used for a variety of reasons, but because this lesson refers to enumerations for the first time in this book, a little sidebar explanation seems appropriate.

If you need to define a group of predefined constants, you can use the `final` keyword. For example, to define various categories of products in a store you can declare the following constants:

```
public static final int CATEGORY_BOOKS  = 1;
public static final int CATEGORY _VIDEOS  = 2;
public static final int CATEGORY _ELECTRONICS  = 3;
public static final int CATEGORY _TOYS  = 4;
```

But Java offers a special keyword, `enum`, that enables you to define such constants in a more elegant way:

```
enum Category {BOOKS,VIDEOS,ELECTRONICS,TOYS}
```

Not only is this notation more compact, but it also prevents the programmers from having to prefix each constant with `CATEGORY_` to stress that these values represent a group, but you can define variables to represent the entire group, pass `enum` as an argument to a method, and return `enum` from a method. For example, if you need to write a program that will redirect the store customer to the proper representative, you can write something similar to what's shown in Listing 20-13.


Available for download on Wrox.com

LISTING 20-13: enum example

```
enum Category {BOOKS,VIDEOS,ELECTRONICS,TOYS}

public class EnumExample {
      ...
    public static void redirectToRepresentative(Category cat){

          switch(os) {
            case BOOKS:
                System.out.println("Get Mary on the phone");
                break;
            case VIDEOS:
                System.out.println("Call Raj");
                break;
            case ELECTRONICS:
                System.out.println("Find Boris");
                break;
            case TOYS:
                System.out.println("Email David");
                break;
```

```
                default:
                    System.out.println("We don't sell such
        products.");
                    break;
                }
            }
        }
```

All `enum`s implicitly extend `java.lang.Enum` and implement `Comparable` and `Serializable` interfaces. On top of that you can declare `enum` variables that implement other interfaces, and declare additional methods, for example:

```
enum Category implements Sellable {
     BOOKS,VIDEOS,ELECTRONICS,TOYS;

   printSalesData(int region){
      // Calculate sales here
   }

}
```

THREAD STATES

A thread goes through various states during its life span. The class `Thread` has a method, `getState()`, that will return one of the values defined in the enumeration `Thread.State`.

- `BLOCKED`: Thread state for a thread that is blocked and waiting to enter or reenter a synchronized method or block of code
- `NEW`: Thread state for a thread that has been instantiated but not started yet
- `RUNNABLE`: Thread state for a runnable thread
- `TERMINATED`: Thread state for a terminated thread
- `TIMED_WAITING`: Thread state for a waiting thread with a specified waiting time
- `WAITING`: Thread state for a waiting thread

The class `Thread` has a method, `isAlive()`, that can help you to find out the status of the thread. If it returns `true`, the thread has been started and hasn't died yet. If it returns `false`, the thread is either new or already dead.

WAIT AND NOTIFY

The class `Object` also has some methods relevant to threads: `wait()`, `notify()`, and `notifyAll()`. Because every Java class is inherited from the class `Object`, these methods can be called on any object.

Let's revisit our class `TestThreads3`, which spawns the threads `MarketNews3` and `Portfolio3`. It has the following line at the end of the `main()` method:

```
System.out.println("The main method TestThreads3 is finished");
```

Run the program `TestThreads3` and it'll print on the system console something like this:

```
The stock market is improving 1
You have  500  shares of IBM
The main method of TestThreads3 is finished
The stock market is improving 2
You have  501  shares of IBM
The stock market is improving! 3
You have  502  shares of IBM
...
```

Note that the method `main()` did not wait for the portfolio and market news threads' completion! But if the `main` class needs to wait until the threads complete their actions, you can use the method `wait()`:

```
public static void main(String args[]){
  ...
 mn.start();
  ...
 p.start();

 synchronized (this) {
   try{
     wait(10000);
   } catch (InterruptedException e){  ...}
 }
  System.out.println("The main method of TestThreads3 is finished");
}
```

The method call `wait(10000)` means "wait for up to 10 seconds." The last `println` statement will be executed either after 10 seconds or when this thread receives notification of some important event, whichever comes first. Examples of important events are a price drop on the auction for items you're interested in, the reopening of the airport after freezing rain, and the execution of your order to purchase 100 shares of IBM stock. A thread can notify other thread(s) by calling the method `notify()` or `notifyAll()`.

Calling `sleep(10000)` puts a thread into a not-runnable state for exactly 10 seconds, although `wait(10000)` may mean that it will come back to a runnable state earlier. If the method `wait()` is called without any arguments, the calling program will wait indefinitely until it receives a notification. The call to `wait()` makes the current thread give up its lock to another one and puts it to sleep until either the specified time has elapsed or the thread gets a notification from another object. Let's consider one of the use cases illustrating this.

`ClassA` spawns a thread, `ClassB`, and starts waiting. `ClassB` retrieves some data, and then sends a notification back to the instance of `ClassA`. The `ClassA` resumes its processing after notification has

been received. Listing 20-14 shows implementation of this scenario. Note that the `ClassB` gets the reference to the object `ClassA` so it knows who to notify.



LISTING 20-14: Notification example

```
class ClassA  {
 String marketNews = null;

 void  someMethod(){

   // The ClassB needs a reference to the locked object
   // to be able to notify it

   ClassB myB=new ClassB(this);
   myB.start();
   synchronized(this) {
    wait();
   }

   // Some further processing of the MarketData goes here...
  }

  public void setData (String news){
    marketNews = news;
  }
}

class ClassB extends Thread{

   ClassA parent = null;

   ClassB(ClassA caller){
     parent = caller; // store the reference to the caller
   }

   run(){
    // Get some data, and, when done, notify the parent
    parent.setData("Economy is recovering...");
    ...
    synchronized (parent){
     parent.notify();  //notification of the caller
    }
   }
}
```

The method `notifyAll()` notifies and wakes up *all* threads waiting for the lock to be released on this object. If several threads are waiting, those that have the highest priority will run first. If several waiting threads have the same priority, the order of awakening is decided by the JVM. While `notify()` can work a tiny bit faster than `notifyAll()`, the latter is able to create objects that are *loosely coupled* — meaning that they don't know about each other.

In Listing 20-14 simple replacement of `parent.notify()` with `notifyAll()` is not enough because `ClassB` also passes the data about the economy back to `ClassA`. But if `ClassB` just needs to notify you about some important application event, `notifyAll()` can be a better choice.

TRY IT

Implement and test the example from Listing 20-14 and then rewrite it to reduce the tight object coupling of the objects. `ClassB` shouldn't know the exact type of the object to notify (`ClassA`) — this makes `ClassB` useful with other objects, too. You'll need to introduce a new interface, say `Updatable`, that declares one method, `setData()`, to make `ClassB` work not only with `ClassA`, but with any class that implements `Updatable`.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at `www.wrox.com`*. You can find them in the Lesson20 folder in the download.*

Step-by-Step

1. Create `ClassA` as in the code shown in Listing 20-14, and add the method `main()` to it.
2. Create `ClassB` and test the application to see that `ClassA` receives the notification about the economy.
3. Declare the new interface, `Updatable`.
4. Modify the constructor of `ClassB` to accept any `Updatable` object.
5. Run and test the new version of the application.



Please select Lesson 20 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

21

Digging Deeper into Concurrent Execution

Even in such a concise tutorial as this one, Java threads deserve two dedicated lessons. Let's continue delving deeper into the world of multi-threaded applications.

JOINING THREADS

In Lesson 20 you learned that a thread can wait for a fixed period or for a notification on some event. Now let's consider a scenario in which you need to start multiple threads and continue program execution only when all threads are complete. The `Thread` class has a method, `join()`, that you can use in this case.

Revisit the TestThreads3 program shown in Listing 20-9. If you run this program the system console will show the message "The main method of TestThreads3 is finished"; after that will it show the output of the portfolio and market news threads, which keep running for a while. If you want to make sure that the main method is waiting until the other two threads are finished, you can use the method `join()`, as shown in Listing 21-1.



LISTING 21-1: Joining threads

```
public class TestThreadJoin {

    public static void main(String args[]){

        MarketNews3 mn = new MarketNews3("Market News");
        mn.start();

        Portfolio3 p = new Portfolio3("Portfolio data");
        p.start();

        try{
```

continues

LISTING 20-1 *(continued)*

```
                mn.join();
                p.join();

            }catch (InterruptedException e){
                e.printStackTrace();
            }

           System.out.println( "The main method of TestThreadJoin is finished");

    }
}
```

The output of the program from Listing 21-1 is shown next. Both `Portfolio3` and `MarketNews3` are joined with the main application thread, which prints its message that the main method is finished only after the other two are done.

```
You have 500 shares of IBM
The market is improving 0
You have 501 shares of IBM
The market is improving 1
You have 502 shares of IBM
You have 503 shares of IBM
The market is improving 2
You have 504 shares of IBM
The market is improving 3
You have 505 shares of IBM
You have 506 shares of IBM
The market is improving 4
You have 507 shares of IBM
The market is improving 5
You have 508 shares of IBM
The market is improving 6
You have 509 shares of IBM
The market is improving 7
The market is improving 8
The market is improving 9
The main method of TestThreadJoin is finished
```

Make a small change in the code and place `mn.join()` before the start of the portfolio thread, and you'll see different output — the ten messages about the market improving will be printed first, followed by the ten messages about your shares of IBM.

GOODIES FROM JAVA.UTIL.CONCURRENT

Java 5 introduced lots of goodies that make thread programming a lot more robust and flexible, and most importantly that increase the performance of multi-threaded applications. In this section I'll just highlight some of the must-know techniques, classes, and interfaces from this package. For detailed coverage of this subject get the book *Java Concurrency in Practice*, Brian Goetz et al., 2006 Addison-Wesley, ISBN: 0321349601.

ReentrantLock versus Synchronized

The package `java.util.concurrent.locks` includes the class `ReentrantLock`, which can be used as a replacement for the `synchronized` keyword. Using it may improve the performance of your code. The idea is to place a lock before the section of your program that may cause a race condition, and to remove the lock afterward. The next code snippet is a revision of the code shown in Listing 20-12:

```
private Lock accountLock = new ReentrantLock();

witdrawCash(int accountID, int amount){
    // Some thread-safe code goes here, i.e. reading from
    // a file or a database
   ...

   accountLock.lock(); // place a lock when this thread enters the code

   try{
   if (allowTransaction){
    updateBalance(accountID, amount, "Withdraw");
   }
   else {
    System.out.println("Not  enough money on the account");
   }
  }finally {
     accountLock.unlock(); // allow other threads to update balance
  }
 }
```

Note that the lock has to be removed in the `finally` section to ensure that unlocking always gets executed, even if there is an exception thrown from the `try` block. When the code is unlocked it can be given to one of the waiting threads. The class `ReentrantLock` has an overloaded constructor with a `boolean` argument — if you specify `true` while creating the lock, the control will be given to the longest-waiting thread.

There is another useful class, `Condition`, that can be associated with the lock. This object enables a locked block of code to suspend its execution until other threads notify the current one that some condition has become `true`, e.g., the bank account has enough funds now for you to make a withdrawal.

If you don't need the flexibility offered by the `ReentrantLock/Condition` combo, just use the `synchronized` keyword with `notify()/notifyAll()` methods to control thread locking. Or, even better, see if using one of the concurrent collections (reviewed in the section "A Brief Introduction to Concurrent Collections") can take care of all your locking needs so you don't need to create explicit locks in your code.

Executor Framework

Creating threads by subclassing `Thread` or implementing `Runnable` works, but there are certain shortcomings to these approaches. First, the method `run()` cannot return a value. Second, an

application may spawn so many threads that it can take up all the system resources, and if this happens the application will stop functioning. In other words, you need to control the number of threads allowed for each application.

You can overcome the first shortcoming by using the `Callable` interface, and the second one by using classes from the *Executor framework*. The `Executors` class spawns the threads from `Runnable` objects, `ExecutorService` knows how to create `Callable` threads, and `ScheduledExecutorService` allows you to schedule threads for future execution.

The utility class `Executors` has static methods that will enable you to create an appropriate executor. In particular, its method `newFixedThreadPool()` creates a pool of threads of a specified size. For example, `Executors.newFixedThreadPool(5)` gives you an instance of `ExecutorService` that automatically supports a pool of not more than five threads. If all five threads are busy when a request to create a new thread comes in, that request will wait until one of the running threads completes. Using thread pools ensures that you can control system resources better.

If you need a thread to return some data on completion, create a class that implements the `Callable` interface and defines a method `call()` that plays the same role as `run()` in `Runnable`. In this case you'll need to create threads differently; the class `Thread` won't take a `Callable` object as an argument. The class `Executors` comes to the rescue: it offers a number of static methods that will create a thread from your `Callable` class and return the result of its execution packaged inside the special object implementing the interface `Future`.

The method `call()` is defined with a parameterized value (remember generics?):

```
public interface Callable <V>{
   V call() throws Exception;
}
```

Accordingly, if some method needs to create a thread using `Callable`, the code should instantiate the `Callable` thread with a specific data type in place of `<V>`. For example, the thread `Portfolio` may return an `Integer` as a result of some processing in its `call()` method:

```
public class Portfolio implements Callable<Integer>{

   public Integer call() {
      // Perform some actions
      return someInteger;

  }
}

public class MarketData implements Callable<Integer>{

   public Integer call() {
      // Perform some actions
      return someInteger;

  }
}
```

Page 262 of 506 (IPR2025-00350)
Stingray Group Inc. et al v. Hernandez-Mondragon

One way to create a `Future` object is by submitting an instance of the `Callable` thread to the `Executor`. Call the function `get()` on the `Future` instance, and it'll block on the thread until its `call()` method returns the result:

```
//Threads' results can be stored in the collection of Futures
List<Future<Integer>>  threadResults= new ArrayList<Future<Integer>>();

// Submit Callables for execution
threadResults.add(myExecutorService.submit(new Portfolio()));
threadResults.add(myExecutorService.submit(new MarketData()));

for (Future<Integer> future : threadResults) {
       future.get();
}
```

Calling methods `get()` on several instances of the `Future` objects is equivalent to joining threads.

The process of spawning threads using `Executors`, `Callable`, and `Future` may go like this:

1. Declare and instantiate a class that implements the `Callable` interface, and program the business logic in its method `call()`.
2. Create an instance of the `Future` object.
3. Create an instance of an `ExecutorService` using `Executors.newFixedThreadPool()`.
4. Call the function `submit()` on the `ExecutorService`, providing an instance of the `Callable` object as an argument.
5. Call the function `get()` on the `Future` object from Step 2. This function will wait until the thread returns the result (or throws an exception).
6. Accept the result of the thread execution into a variable of the data type used in Step 1.
7. Call the function `shutdown()` on the `ExecutorService` from Step 3.

The code in Listings 21-2 to 21-4 demonstrates the spawning of the familiar threads portfolio and market news using the classes from the Executor Framework. In this example each `Callable` thread performs some dummy actions and returns an integer as a result.



LISTING 21-2: Portfolio that implements Callable

```
class PortfolioCallable implements Callable<Integer> {

  public Integer call() throws Exception {
   for (int i=0; i<5;i++){
       Thread.sleep (700);    // Sleep for 700 milliseconds
       System.out.println( "You have " +  (500 + i) +
                                      " shares of IBM");
   }

   // Just return some number as a result
```

continues

LISTING 21-2 *(continued)*

```
    return 10;
   }
  }
```



LISTING 21-3: MarketNews that implements Callable

```
class MarketNewsCallable implements Callable<Integer> {

  public Integer call() throws Exception {

         for (int i=0; i<5;i++){
             Thread.sleep (1000);  // sleep for 1 second
              System.out.println( "The market is improving " + i);
            }
    // Just return some number as a result
    return 12345;
   }
  }
```

The `TestCallableThreads` class creates a collection of `Future` objects — one per thread. Executor creates a pool of two threads and each thread is submitted for execution. The method `get()` waits for the completion of each thread, and the result of each `call()` method is stored in the collection `results`.



LISTING 21-4: Spawning threads with the Executor framework

```
public class TestCallableThreads {

  public static void main(String[] args)
   throws InterruptedException, ExecutionException {

   //A placeholder for Future objects
       List<Future<Integer>> futures =
     new ArrayList<Future<Integer>>();

    // A placeholder for results
       List<Integer> results = new ArrayList<Integer>();

   final ExecutorService service =
     Executors.newFixedThreadPool(2); // pool of 2 threads

   try {

    futures.add(service.submit(new PortfolioCallable()));
    futures.add(service.submit(new MarketNewsCallable()));

    for (Future<Integer> future : futures) {
     results.add(future.get());
```

```
        }

       } finally {
        service.shutdown();
       }

       for (Integer res: results){
          System.out.println("\nGot the result: " + res);
       }
      }
     }
```

The output of this program is shown next. But if you change the number of threads in the pool from two to one, the program will first print all messages from the portfolio thread and only after that print all messages from the market news.

```
You have 500 shares of IBM
The market is improving 0
You have 501 shares of IBM
The market is improving 1
You have 502 shares of IBM
You have 503 shares of IBM
The market is improving 2
You have 504 shares of IBM
The market is improving 3
The market is improving 4

Got the result: 10
Got the result: 12345
```

A Brief Introduction to Concurrent Collections

The package `java.util.concurrent` offers a number of data structures that simplify programming with threads. I'll just briefly name some of them in this section.

Queues

The concept of a *queue* (First In First Out or FIFO) fits well in any process that involves asynchronous intra-object communications. Instead of object `A` trying to place a direct lock on object `B`, the former (aka the *producer*) can place some data objects in a queue, and the latter (aka the *consumer*) will retrieve (*dequeue*) them from the queue asynchronously. Most importantly, the queues from the `java.util.concurrent` package are *thread-safe*, which means that you can add an object to a queue without worrying about race conditions.

If the queue is blocking, the thread will also block while trying to add an object to a full queue or remove an object from an empty one. The following classes implement the `BlockingQueue` interface: `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`, and `DelayQueue`. To add an object to a queue you can use such methods as `add()`, `put()`, and `offer()`. To retrieve an object from a queue use `poll()`, `remove()`, `take()`, or `peek()`.

Unbound queues don't place limitations on the number of elements. `ConcurrentLinkedQueue` is an example of such a queue.

Java 6 has introduced a `Deque` interface for inserting and removing elements from both ends of the queue. The class `LinkedBlockingDeque` is a concurrent implementation of this interface.

Collections

Using concurrent collections is a recommended way of creating thread-safe data structures. Such collections include `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`. Java documentation describes when to use each of these collections. For example, a `CopyOnWriteArrayList` is preferable to a `synchronized ArrayList` when the expected number of reads and traversals is much greater than the number of updates to a list. These collections were written to minimize the time during which data is locked.

The utility class `java.util.Collections` has a number of static methods that create thread-safe collections. Their method names start with the word *synchronized*. For example, `synchronizedList()` takes a regular `List` (such as `ArrayList`) as an argument and makes it thread-safe. You can read more about Java collections at `http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html`.

Finding a ready-to-use synchronized collection is better than writing `synchronized` blocks on your own — the chances are slim that you'll write more efficient synchronization code than already exists in Java.

SWINGWORKER THREAD

Any Java Swing application spawns a number of threads. At the very minimum it runs the main application thread, the second one captures system events, and the third communicates with the GUI. The application itself may spawn additional threads. But if more than one thread will need to update the UI components, the changes may not be rendered properly, because Swing components were not made thread-safe to minimize the number of locks that hurt performance.

To avoid this problem, UI updates shouldn't be made directly, but rather submitted to an *event dispatch thread*. Swing uses a single-threaded model, which means that all UI updates are rendered via a single thread.

Suppose your GUI application is written with Java Swing, and a click on `JButton` initiates some server-side data request that takes about 10 seconds to complete. You should never execute long requests in the event dispatch thread. If you do, then the UI will become frozen, as no updates can be made until the long running process releases the lock on the thread. Therefore, you need to start a separate thread for such a long process, and when it finishes, the program has to modify the GUI via the event dispatch thread.

For example, if the result of a button click has to update a `JTextField`, you may create a new thread in the button's `actionPerformed()` method and, from within the `run()` method of this thread, update the text field. This will work…most of the time, if there are no conflicts with other threads running in your application.

All UI-related Swing events (such as button clicks and window repaints) are placed in a special queue, and the object `java.awt.EventQueue` retrieves them from this queue. You should direct modification of the UI (the `JTextField` in our example) to this queue.

In the older version of Java, to ensure that all application-specific data would modify the GUI via this queue, developers used the method `invokeLater()` to ensure that UI changes were placed in the `EventQueue`:

```
SwingUtilities.invokeLater()
   new Runnable(){
    public void run(){
      // Do some processing here
      //... and then update the UI
      myTextField.setText(someData);
    }
  }
);
```

While this technique still works, it clogs the code with anonymous `Runnable` classes. The class `javax.swing.SwingWorker` gives you a cleaner (though not necessarily simpler) means of dealing with the event dispatch thread. This thread class implements `Runnable` and `Future`, and so can be submitted to the `Executor` for execution and return a result.

Let's say a Swing application needs to make a request to the server to get the market news information, which may take a couple of seconds. So that the UI won't be frozen for these seconds, this request has to be performed in a background thread, and when the result is ready the UI has to be updated. To arrange this, create a subclass of `SwingWorker` and override its `doInBackground()` method, then instantiate it and call its `execute()` method, as in Listing 21-5.



LISTING 21-5: Basic use of SwingWorker

```
class MarketNewsWorker extends SwingWorker <List<String>, String>{

   @Override public List<String> doInBackground(){
       // Make a request to the server and return a result,
       // i.e. a list of Strings
      return myListOfTextData;
  }
     // method method overides go here
 }

class TestMarketNews{
     ...
     public static void main(String[] args){
          new MarketNewsWorker().execute();
     }
}
```

This code gives you a high-level picture, but there is more to executing the thread with `SwingWorker`. First, you probably noticed the unknown syntax element `@Override`, which is Java *annotation* stating that the method `doInBackground()` is being overridden. Adding the `@Override` annotation is not required here; it's just an example of an annotation. You'll learn about annotations in Lesson 24.

Second, the class `MarketNewsWorker` uses generics and has two parameters, `<List<String>` and `String>`. The reason for this is that the overridden method `doInBackground()` *might* call the `SwingWorker`'s `process()` method, and *will* call its `done()` method on completion — this is where the UI is being updated. Two parameters indicate what types of data will be returned by `doInBackground()` and given to `process()` respectively.

Why might you consider calling the method `process()` during your thread execution? You might do it to support some kind of progress meter or other means of reporting the progress of long-running processes. If, for example, a long-running thread is reading a large file or performing some lengthy calculations, you might want to calculate the percentage of completion and report it to the calling Swing application.

You are not allowed to call the `process()` method directly, but have to call a method called `publish()`, which will internally call `process()`. Override `process()` to add some messages to the log file or update the progress meter. The code to display the result of the calculations on the UI should be written in the method `done()`.

Listing 21-6 shows you a typical way to program with `SwingWorker`. I left out the details on purpose so you'd have something to do for today's homework.


Available for download on Wrox.com

LISTING 21-6: A typical way to use SwingWorker

```
class MarketNewsWorker extends SwingWorker <List<String>, String>{

   @Override public List<String> doInBackground(){
       // Make a request to the server and return a result,
       // i.e. a list of Strings
       for (String news: someNewsCollection){
            //process each news and report the progress
             ...
            publish("Processed the news " + news); //this calls process()
       }
      return myListOfTextData;
  }

   @Override protected void process(String progressMessage){
       // display the progress information here
   }

   @Override protected void done(){
       // modify UI components here by calling get()
       // Future's get() gives you the result of
       // the thread execution
   }
}

class TestMarketNews{
     ...
     public static void main(String[] args){
          new MarketNewsWorker().execute();
     }
}
```

You just completed a rather advanced lesson of this book. The subject definitely requires more research and practice. Some good content to read next is the lesson on concurrency in Oracle's Java tutorial, which you'll find here: `http://download-llnw.oracle.com/javase/tutorial/essential/concurrency/`. As always, trying it hands-on will deepen your understanding.

TRY IT

Create a Swing application with the UI that consists of two `JTextArea` controls and one `JButton` with the label "Get the News." Prepare two text files with some text information (the news), and write the code that reads them to the left and right text areas respectively. File reading has to be implemented concurrently using two `SwingWorker` threads.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at `www.wrox.com`. *You can find them in the Lesson21 folder in the download.*

Step-by-Step

1. Create a new Eclipse project called Lesson21.
2. Create a class called `NewsReader` as a subclass of `SwingWorker`. This class should have a constructor that takes one argument of type `File`.
3. Prepare two text files with some news in each.
4. Create a Swing application with two text areas and a button.
5. On the click of the button instantiate two `NewsReader` threads. Each thread should get an instance of the `File` object pointing to the corresponding news file.
6. Override the `NewsReader`'s methods `doInBackground()` and `done()` to read the files and populate the Swing view.
7. Test this program.
8. This step is optional. Override the method `process()` and make sure that it updates the view with progress information about the reading process. The progress should be displayed as a percentage: The percentage formula is `progressToDisplay=readBytes/FileSize*100`.



Please select Lesson 21 on the DVD with the print book, or watch online at `www.wrox.com/go/fainjava` *to view the video that accompanies this lesson.*

22

Working with Databases Using JDBC

Business applications usually store data in the databases. In most of the enterprise applications, Relational Database Management Systems (RDBMSes) are used as data storage. They store the data records in *tables*. Each record (such as that of an employee) is represented by a table row, which consists of one or more columns or *record fields* (e.g., name, address, hire date). RDBMSes understand the SQL language.

The most popular RDBMSes are Oracle, DB2, Sybase, Microsoft SQL Server, and MySQL Server. All samples from this lesson will use JavaDB, which is included with your JDK installation.

Java includes two packages that contain classes required for work with DBMSes: `java.sql` and `javax.sql`. The former contains commonly used classes such as `Connection`, `Statement`, and `ResultSet`. The latter is used for supporting database connectivity on the server side, containing classes such as `DataSource`, and `RowSet`.

The JDBC API is not DBMS-specific — if you write a program that uses JDBC classes to retrieve/update data in Oracle, you'll be using the same classes to work with MySQL Server or DB2. You just need the JDBC drivers from the corresponding DBMS vendor — the drivers hide their database specifics behind the same public JDBC API.

JDBC drivers either pass SQL statements from Java to a DBMS for execution or simply execute a program stored inside a DBMS (called a *stored procedure*). If some data has been retrieved as the result of these actions, your Java program will handle it by making appropriate calls to the JDBC API. Over the past 15 years the JDBC specification has been evolving and, at the time of this writing, most drivers comply with JDBC version 4.0.

In this lesson all communications with the DBMS are made by supplying SQL statements to the JDBC API. There is an alternative way of working with data, by using one of the object-relational mapping (ORM) frameworks. In Lesson 36 you'll get familiar with a popular ORM framework called Hibernate.

JDBC DRIVER TYPES

A JDBC driver plays the role of the middleman between a Java program and a DBMS. Drivers are available from database vendors, from Oracle, and from third-party vendors of Java application servers.

There are four general types of JDBC drivers.

A Type 1 driver is a JDBC-ODBC bridge that enables Java programs to work with the database using ODBC drivers from Microsoft. The drawbacks of ODBC drivers are that they are slower than the others, must be installed and configured on each user's machine, and work only on Windows machines. The Type 1 JDBC drivers are rarely used.

A Type 2 driver consists of Java classes that work in conjunction with the non-Java native drivers provided by the DBMS vendor. These drivers work much faster than Type 1, but also require installation and configuration on the machine on which Java programs run.

A Type 3 driver is called a *middleware driver* and can be provided by vendors of application servers. It consists of two parts: the *client* portion performs a DBMS-independent SQL call, which is then translated to a specific DBMS protocol by the *server* portion of the driver.

A Type 4 driver is a pure Java driver, which usually comes as a `.jar` file and performs direct calls to the database server. It does not need any configuration on the client's machine, other than including the name of the main driver's class in your Java code. That's why it's also known as the *thin driver.* For example, Java applets can be packaged with this type of driver, which can be automatically downloaded to the user's machine along with the applets themselves.

For simplicity I'll be using JDBC drivers of Type 4 in this lesson, but many production systems can deploy Type 3 drivers to provide better performance.

CREATING A DATABASE WITH DERBY

In this lesson I'll be using an open-source DBMS called JavaDB (also known as Derby), which is packaged with Java 6. It's a small DBMS that you wouldn't use for serious production systems, but it's great for learning JDBC or to use for many small systems. If you've installed Java SDK as instructed in Lesson 1 (see Figure 1-2), you already have JavaDB installed in the folder `c:\Program Files\Sun\JavaDB`. Modify your system variable `PATH` so it starts with the following code:

```
c:\Program Files\Sun\JavaDB\bin;
```

To make sure that you change `PATH` properly, open a command window and enter `sysinfo`. This command should print information about your Java environment, and you should find a reference to Derby and JDBC 4.0, as shown in Figure 22-1.

If you don't have JavaDB installed, download and install it from `http://db.apache.org/derby`. Derby is well documented and if you haven't had a chance to work with relational DBMSes, download and read the "Getting Started with Derby" manual.

Derby has an interactive command-line utility called ij that you can use to create databases and tables and populate them with data, among other actions. I'll show you how to create a sample

database and a table to store data about employees. First open a command window and issue the command `startNetworkServer`. Figure 22-2 shows the message that Derby has started and is ready to accept connections on port 1527.

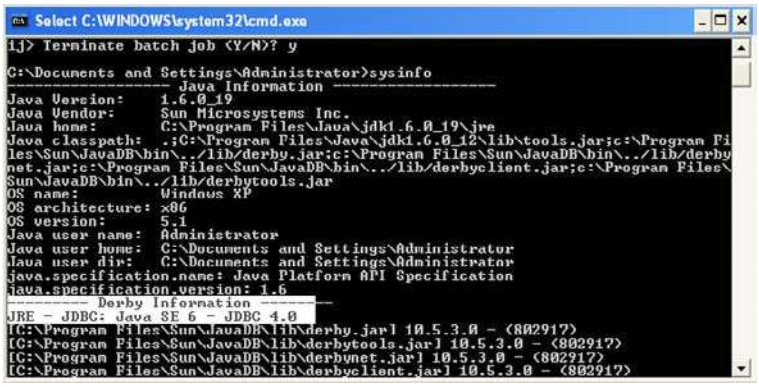


FIGURE 22-1

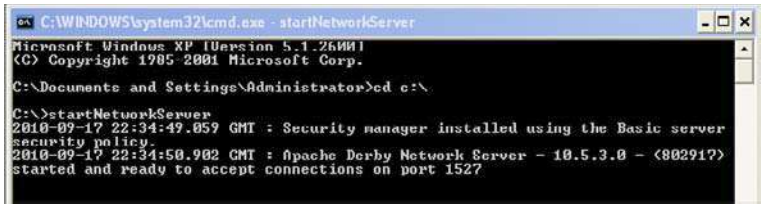


FIGURE 22-2

Open another command window and start ij. First, connect to the database Lesson22 by issuing the following command:

```
connect 'jdbc:derby://localhost:1527/Lesson22;create=true';
```

This command will try to connect to the database Lesson22, and will create it if no such database is found. The next command will create a database table, Employee, to store records that consist of three fields: `EMPNO`, `ENAME`, and `JOB_TITLE`. The first field will be stored as an integer, and the other two as simple text (varchars) allowing 50 and 150 characters respectively.

```
CREATE TABLE Employee (
    EMPNO int NOT NULL,
    ENAME varchar (50) NOT NULL,
    JOB_TITLE varchar (150) NOT NULL
);
```

Finally, to populate the table with some data, issue the `INSERT` command in ij:

```
INSERT INTO Employee values (7369,'John Smith', 'Clerk'), (7499,
'Joe Allen','Salesman'), (7521,'Mary Lou','Director');
```

Figure 22-3 is a snapshot of the command window after the database Lesson22 and the table Employee have been created, and the three records have been inserted there.

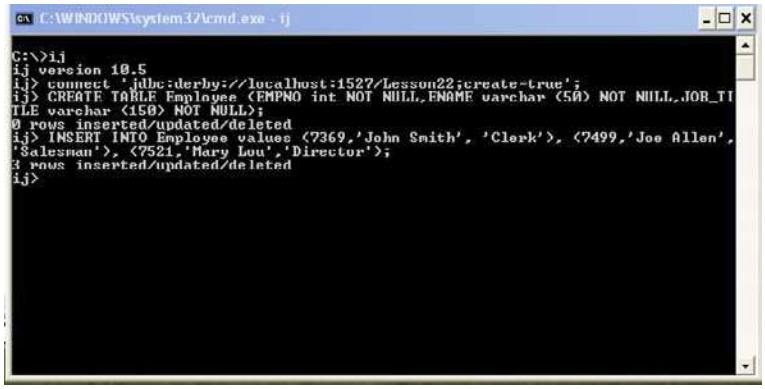


FIGURE 22-3

If you want to ensure that the records were successfully created in the database, issue the `SELECT` SQL statement to retrieve the data:

```
Select * from Employee;
```

You'll see the output of the above SQL statement in the next section. If you are not familiar with the syntax of SQL, refer to the tutorial at `www.w3schools.com/sql/default.asp`.

SAMPLE JDBC PROGRAM

In this section I'll show you the steps that you can perform to retrieve the data in any Java program that works with a relational database using JDBC. A sample program will implement all of these steps to display the list of employees from the database table Employee.

1. Load the JDBC driver using the method `forName()` of the Java class `Class`. You have to find out the name of the class to load from the JDBC driver's documentation. If you work with Oracle DBMSes, you can load a Type 4 JDBC driver with the following command:

   ```
   Class.forName("oracle.jdbc.driver.OracleDriver");
   ```

   In the case of JavaDB, which was installed and registered with your JDK, you can skip this step.

2. Obtain the database connection to the database Lesson22 by calling

   ```
   DriverManager.getConnection(url, user, password);
   ```

   In the case of Derby DB you don't have to supply the user and the password; simply provide the URL of your database, for example

   ```
   DriverManager.getConnection("jdbc:derby:Lesson22");
   ```

3. Create an instance of the Java class `Statement`:

```
Connection.createStatement();
```

As an alternative, you can create `PreparedStatement` or `CallableStatement`, explained later in this lesson in "The PreparedStatement Class" and "The CallableStatement Class" sections later in this lesson.

4. For SQL `Select` execute the following command:

```
Statement.executeQuery("Select ...");
```

For SQL queries, which produce more than one result, use the method `execute()`.

For `Insert`, `Update`, and `Delete` SQL statements use the following:

```
Statement.updateQuery("Insert ...");
```

5. Write a loop to process the database result set, if any:

```
while (ResultSet.next()) {...}
```

6. Free system resources by closing the `ResultSet`, `Statement`, and `Connection` objects.

All these steps are implemented in the class `EmployeeList`, shown in Listing 22-1, which prints the records from the table Employee. Even though you don't need to explicitly load the driver for Derby with `Class.forName()`, the location of the driver class has to be known to your program, otherwise you'll get a "No suitable driver" error. Either add `derbyclient.jar` to the `CLASSPATH` system variable, or just add it as an external `.jar` to your Eclipse project (see the Java Build Panel, available from the Eclipse project Properties menu).



LISTING 22-1: The EmployeeList program

```
package com.practicaljava.lesson22;


import java.sql.*;
class EmployeeList {
public static void main(String argv[]) {

   Connection conn=null;
   Statement stmt=null;
   ResultSet rs=null;

  try {

    // Load the JDBC driver - Class
    // This can be skipped for Derby, but derbyclient.jar
```

continues

LISTING 22-1 *(continued)*

```
    //has to be in the CLASSPATH

    // Class.forName("org.apache.derby.jdbc.ClientDriver");

    conn = DriverManager.getConnection("jdbc:derby://localhost:1527/Lesson22");


    // Create an SQL query
    String sqlQuery = "SELECT * from Employee";

    // Create an instance of the Statement object
    stmt = conn.createStatement();

    // Execute SQL and get obtain the ResultSet object
    rs = stmt.executeQuery(sqlQuery);


    // Process the result set - print Employees
    while (rs.next()){
    int empNo = rs.getInt("EMPNO");
        String eName = rs.getString("ENAME");
        String job = rs.getString("JOB_TITLE");

System.out.println(""+ empNo + ", " + eName + ", " + job );
    }
   } catch( SQLException se ) {
      System.out.println ("SQLError: " + se.getMessage ()
           + " code: " + se.getErrorCode());
   } catch( Exception e ) {

      System.out.println(e.getMessage());
      e.printStackTrace();
   } finally{

       // clean up the system resources
       try{
rs.close();
stmt.close();
conn.close();
       } catch(Exception e){
           e.printStackTrace();
       }
   }
 }
}
```

The output of the program EmployeeList should look like this:

```
7369, John Smith, CLERK
7499, Joe Allen, SALESMAN
7521, Mary Lou, Director
```

When you execute any SQL statements, always include processing of errors that may be returned by the DBMS. Catching the `SQLException` is the right way to get the error message. Note that the code in Listing 22-1 calls the method `getErrorCode()` to extract the database-specific error code from the `SQLException` object.

PROCESSING RESULT SETS

Let's take a closer look at the code in Listing 22-1. After `rs = stmt.executeQuery(sqlQuery)`, the cursor `rs` points at the very first record (row) of the result set in memory. Each row contains as many fields (columns) as were specified in the SQL `Select` statement. Each of the values is extracted by an appropriate method based on the data type of the field. The names of these methods are self-explanatory: `rs.getString()`, `rs.getInt()`, and so on. If you know the name of a column from the result, use it as a method argument:

```
int empNo = rs.getInt("EMPNO");
String eName = rs.getString("ENAME");
```

JDBC drivers are smart enough to convert the data from the database types to the corresponding Java types: For example, Derby's `varchar` will become Java's `String`.

You could have gotten the values of each field by specifying the relative position of the column from the result set:

```
int empNo = rs.getInt(1);
String eName = rs.getString(2);
```

Besides having names, columns in the result set are numbered starting with `1`. In some cases numbers are your only choice; for example, the following SQL query does not list column names to select (the asterisk means "all"):

```
stmt.executeQuery("Select count(*) from EMP");
```

The class `EmployeeList` just prints the retrieved data in a loop. You can also place the result set in a Java collection object for further processing. The `ResultSet` object holds the database connection and is not serializable. That's why common practice for programming server-side operations with DBMSes is to create a class representing a row from the result set and populate, say, an `ArrayList` or other Java collection with its instances.

Listing 22-2 shows an example of such a class, which can represent one employee record. Classes that hold only the value of some data are often called *value objects*. Because in distributed applications such objects may need to be transferred between different computers, they are also known as *Data Transfer Objects (DTOs)*.



LISTING 22-2: The EmployeeDTO

```
class EmployeeDTO{

  //private properties
  private int empNo;
  private String eName;
  private String jobTitle;

  //setters
  public void setEmpNo(int val){empNo=val;}
  public void setEName(String val){eName=val;}
  public void setJobTitle(String val){jobTitle=val;}

  // getters
  public int getEmpNo(){return empNo;}
  public String getEName(){return eName;}
  public String getJob(){return jobTitle;}
}
```

`EmployeeDTO` declares private variables to store the data but access to this data is performed via public *setters* and *getters*, the methods that allow external code to set and get the appropriate values. This technique can be useful when some application-specific logic has to be applied at the moment when some code needs to get or modify the properties of the class `EmployeeDTO`.

For example, you can place some authorization code inside the setter to ensure that the external object has enough permissions to change the property `jobTitle`. If the business logic of obtaining such authorization changes in the future, you need to modify only the code inside the setter, but the external code will remain unchanged.

The next code snippet shows how the class `EmployeeList` can prepare a collection of `EmployeeDTO` objects while processing the result set retrieved by the SQL `Select` statement.

```
class EmployeeList {
...
 // Create an object for collection of employees
 ArrayList<EmployeeDTO> employees = new ArrayList<EmployeeDTO>();

 // Process ResultSet and populate the collection


 while (rs.next()){
  EmployeeDTO currentEmp = new EmployeeDTO();
  currentEmp.setEmpNo(rs.getInt("EMPNO"));
  currentEmp.setEName(rs.getString("ENAME"));
  currentEmp.setJob(rs.getString("JOB_TITLE"));

  employees.add(currEmp);
 }
...
}
```

If this code is deployed on the server's JVM and you need to send the data to another computer that runs, say, a Swing client, you can consider applying Java serialization here for sending a collection of employees to the front. But make sure that the class `EmployeeDTO` implements the `Serializable` interface.

THE PREPAREDSTATEMENT CLASS

Listing 22-1 uses the class `Statement` to create an object capable of executing SQL. But this is not the only way to supply SQL to the JDBC API. The class `PreparedStatement` is a subclass of `Statement`, but it pre-compiles the SQL statement before executing it.

With `PreparedStatement` you can create SQL with parameters that are dynamically passed by the program. Suppose you need to execute the query `"SELECT * from EMP WHERE empno=..."` multiple times, providing the `empno` values from the array `empNumbers[]`. If you use the class `Statement`, the variable `sqlQuery` will have to be pre-compiled on each iteration of the loop:

```
for (int i=0;i<empNumbers.length; i++){
  sqlQuery="SELECT * from Employee WHERE empno=" + employees[i];
  stmt.executeQuery(sqlQuery);
}
```

The class `PreparedStatement` offers a more efficient solution:

```
PreparedStatement stmt=conn.prepareStatement(
                       " SELECT * from Employee WHERE empno=?");

for (int i=0;i<empNumbers.length; i++){

  // pass the array's value that substitutes the question mark
  stmt.setInt(1,employees[i];)
  stmt.executeQuery(sqlQuery);
}
```

In this case the SQL statement is compiled only once and parameters are provided by the appropriate `setXXX()` method depending on the data type. The SQL statement may have several parameters (question marks), and the first argument of the setter enables you to specify each parameter's number. For example:

```
PreparedStatement stmt=conn.prepareStatement(
      "SELECT * from Employee WHERE empno=? and ename=?");

for (int i=0;i<empNumbers.length; i++){
  stmt.setInt(1,empNumbers[i];)
  stmt.setString(2,empNames[i];)
  stmt.executeQuery(sqlQuery);
}
```

If you need to pass a `NULL` value as a parameter, use the method `setNull()`.

THE CALLABLESTATEMENT CLASS

This class extends `PreparedStatement` and is used for executing stored procedures from Java. Let's say there is a stored procedure entitled `changeEmpTitle` that takes two parameters: `empno` and `title`. Here's the code to execute this stored procedure:

```
CallableStatement stmt = conn.prepareCall("{call changeEmpTitle(?,?)}");

stmt.setInt(1,7566);
stmt.setString (2,"Salesman");
stmt.executeUpdate();
```

If a stored procedure returns some values using output parameters, each of the `OUT` data types has to be registered before the statement is executed. The next code snippet shows you an example of executing a stored procedure that has two parameters: The first is an input parameter, and the second is an output parameter by which the stored procedure can return the result of its execution to the Java program:

```
CallableStatement stmt = conn.prepareCall(
                 ("{call getEmpTitle(?,?) }");
stmt.setInt(1, 7566);
stmt.registerOutParameter(2,java.sql.Types.VARCHAR);
stmt.executeQuery();
String title=stmt.getString(2);
```

THE RESULTSETMETADATA CLASS

JDBC enables you to process result sets when the number of returned values is unknown. Imagine that you need to write a program that can accept any SQL `Select` statement, execute it, and display the retrieved data. With the class `ResultSetMetaData` you can dynamically find out how many columns there are in the result set, as well as their names and data types. The following code fragment gets the number of the database table columns in the result set and for each of them identifies and prints the column name and type:

```
String sqlQuery = "select * from Employee";
ResultSet rs = stmt.executeQuery(query);

ResultSetMetaData rsMeta = rs.getMetaData();
int colCount = rsMeta.getColumnCount();

for (int i = 1; i <= colCount; i++)  {
  System.out.println(
       " Column name: " + rsMeta.getColumnName(i) +
       " Column type: " + rsMeta.getColumnTypeName(i));
}
```

This simple but powerful technique is used internally by ORM frameworks (see Lesson 36) that can "magically" generate database models and automatically generate Java classes representing database entities.

Listing 22-3 shows a Java program called ShowAnyData that prints a result set based on any SQL `Select` statement passed from the command line. For example, it can be started as follows:

```
java ShowAnyData "Select * from Employee"
```



LISTING 22-3: Using ResultSetMetaData

```
import java.sql.*;
class ShowAnyData {

 public static void main(String args[]) {
   Connection conn=null;
   Statement stmt=null;
   ResultSet rs=null;

 if (args.length==0){

    System.out.println("Usage: java ShowAnyData SQLSelectStatement");
    System.out.println("For example: java ShowAnyData \"Select * from Employee\"");
    System.exit(1);
  }

   try {
                  // Class.forName("org.apache.derby.jdbc.ClientDriver");
      conn = DriverManager.getConnection(
                           "jdbc:derby://localhost:1527/Lesson22");

      stmt = conn.createStatement();

    // Execute SQL query passed from command line
    rs = stmt.executeQuery(args[0]);

    // Find out the number of columns, their names and display the data
    ResultSetMetaData rsMeta = rs.getMetaData();
    int colCount = rsMeta.getColumnCount();

     for (int i = 1; i <= colCount; i++)  {
      System.out.print(rsMeta.getColumnName(i) + " ");
    }

    System.out.println();

    while (rs.next()){
       for (int i = 1; i <= colCount; i++)  {
         System.out.print(rs.getString(i) + " ");
       }
       System.out.print("\n");   // new line character
    }

   } catch( SQLException se ) {
      System.out.println ("SQLError: " + se.getMessage ()
           + " code: " + se.getErrorCode ());
   } catch( Exception e ) {
      System.out.println(e.getMessage());
```

continues

LISTING 22-3 *(continued)*

```
            e.printStackTrace();
        } finally{

            // clean up the system resources
            try{
                        rs.close();
                        stmt.close();
                        conn.close();
            } catch(Exception e){
                e.printStackTrace();
            }
        }
     }
    }
```

The output of the ShowAnyData program will be the same as that of EmployeeList shown in Listing 22-1. But this program can execute any SQL statement as long as you are specifying valid database objects. Note that the code in ShowAnyData first ensures that you have passed the command-line argument. If you run this program from a command line, don't forget to include the SQL statement in double quotes. In Eclipse you can specify a command-line argument by selecting the Run Configuration panel in the Arguments tab.

SCROLLABLE RESULT SETS AND ROWSET

In all the preceding examples the code traversed the result set using the method `next()`, which moves only the cursor forward. Another option is to create a scrollable result set so the cursor can be moved back and forth if need be. There is a two-argument version of the method `createStatement()`. The first argument specifies the type of scrolling (`TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE`) and the second makes the result set updateable or read-only (`CONCUR_READ_ONLY` or `CONCUR_UPDATABLE`). For example,

```
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * from Employee");
```

The `TYPE_FORWARD_ONLY` parameter allows only forward movement of the cursor. The difference between `TYPE_SCROLL_INSENSITIVE` and `TYPE_SCROLL_SENSITIVE` is in whether scrolling reflects changes that have been made to the result set. The next example sets the cursor at the end of the result set and moves the cursor backward:

```
rs.afterLast();
while (rs.previous()){
 int empNo = rs.getInt("EMPNO");
 String eName = rs.getString("ENAME");
 String job = rs.getString("JOB");
 System.out.println(""+ empNo + ", " + eName + ", " + job_title);
}
```

You can also move the cursor to a specific row by using the following self-explanatory methods:

```
rs.absolute(25);  // moves the cursor to the 25th row
rs.relative(-4);  // moves the cursor to the 21st row
rs.first();
rs.last();
rs.beforeFirst();
```

If the result set is updatable (`CONCUR_UPDATABLE`) you can modify the underlying database table while scrolling. For example, the following statements will update the job title of the employee based on the current cursor's position:

```
rs.updateString("JOB_TITLE","Manager");
rs.updateRow();
```

Scrollable result sets enable you to traverse the result set in both directions, but they have a drawback: They hold the database connection, which may be required by another thread or program. The package `javax.sql` includes the interface `RowSet`, which is a subclass of `ResultSet`. `RowSet` gets the data from the database, then disconnects, but still allows Java to work with the data. The package `javax.sql.rowset` has several concrete classes that implement `RowSet`, such as `CachedRowSet`, `FilteredRowSet`, and `WebRowSet`. The latter can turn `RowSet` into an XML stream to be sent to another tier in the distributed application.

TRANSACTIONAL UPDATES

Sometimes several database modifications have to be processed as one transaction, and if one of the updates fails, the whole transaction has to be *rolled back*. These database operations have to be explicitly *committed* (finalized) in case of success. If you set the auto-commit parameter on the database connection to `false`, the database transaction will not be committed until the code explicitly calls the method `commit()`, as in the following example:

```
try{
  conn.setAutoCommit(false);

  Statement stmt = con.createStatement();

  stmt.addBatch("insert into Orders " +
              "values(123, 'Buy','IBM',200)");
  stmt.addBatch("insert into OrderDetail " +
              "values('JSmith', 'Broker131', '05/20/02')");
  stmt.executeBatch();

  conn.commit();    // Transaction succeded

}catch(SQLException e){
  conn.rollback();  // Transaction failed
  e.printStackTrace();
}
```